Сутність методологій розробки: Від єгипетський пірамід до Agile

Це стаття вводить в суть методологій розробки та робить їх огляд. Вона буде корисна для всіх, хто працює з командами інженерів. Знання методологій розробки входить до переліку знань інженера (SWEBOK від IEEE).

Рік: 2024.

Автор статті Дмитро Попов.
Дмитро Попов сертифікований Scrum-розробник, Software Engineer (ЧНУ) з 10-річним досвідом у сфері розробки програмного забезпечення (працював в Durst Group, Embracer Group, Neatpeak Group).
https://www.linkedin.com/in/dima1popov/

Привіт! Мене звати Дмитро Попов. Протягом кар'єри я працював з різними методологіями, зокрема Waterfall, гнучкими підходами з Kanban, Scrum та Extreme Programming. Хоча я не є проджект-менеджером, моє розуміння методологій ґрунтується на досвіді старшого інженера та лідера команди. Однак цей матеріал буде корисним і для менеджерів. Знання методологій розробки є важливими для програмного інженера, як вважають Роберт Мартін, Кент Бек та Джефф Сазерленд, і ці знання також включені в навчальні плани SWEBOK, ISTQB та визнану книгу "Software Engineering" Роджера Пресмана.

Повчальна історія про важливість оцінки ризиків

Чарльз Беббідж, відомий як батько обчислювальних машин, намагався реалізувати амбітний проєкт різницевої машини, що стала першим механічним пристроєм для автоматичного обчислення значень математичних функцій методом скінченних різниць. (Про цей метод можна прочитати в книзі The Essence of Algorithms by D.Popov)
У 1821 році Чарльз Беббідж отримав державні гроші на створення повноцінної різницевої машини, яку він планував створити за три роки. Але технологія створення машини виявилася дуже складною для того часу, і потрібно було мати тисячі точно виготовлених шестерень, що було дуже дорого.
Спочатку Беббідж співпрацював з механіком Джозефом Клементом, який виготовляв шестерні для його машини, але потім вони перестали працювати через ділові протиріччя. Незважаючи на очікування, до 1833 року була завершена лише частина машини, і Беббідж витратив на її будівництво чимало власних грошей. Він витратив близько £17 000 (з яких £6 000 були його особистими грошима). На ці кошти в той час він міг би придбати близько 20 локомотивів у Джорджа та його сина Роберта Стефенсона.

Хоча Чарльз Беббідж створив кілька часткових прототипів, які продемонстрували потенціал його ідеї, проєкт ніколи не був завершений, але послужив відправною точкою для подальших розробок.

Однак у його роботі можна виявити кілька ключових проблем з точки зору менеджменту, оцінки ризиків і застосування методологій розробки.

По-перше, Беббідж не зміг адекватно спланувати проєкт і оцінити ризики. Технічні можливості того часу були значно обмежені, і навіть хоча він мав план побудови машини, він не врахував складність виготовлення точних механічних частин у великих обсягах. Кошториси були постійно перевищені, а взаємодія з майстрами, які виготовляли деталі, погіршилася через фінансові та організаційні розбіжності. Крім того, під час виконання проєкту Беббідж недооцінив складність управління людьми і стейкхолдерами, що призвело до численних конфліктів і затримок. Враховуючи ці труднощі, Беббідж також не зміг створити прототип, який би довів життєздатність проєкту. Прототипи, що були виготовлені, не відповідали його початковим планам і, хоча й демонстрували основні ідеї, не ставали повноцінними робочими моделями.

По-друге, рішення Беббіджа переключитися з різницевої машини на більш амбіційну аналітичну машину є класичним прикладом ефекту другої системи. Цей ефект, за якого розробники, після першої спроби, прагнуть створити більш складну ідеальну систему, в результаті призводить до розширення проєкту за межі можливого. Беббідж, маючи попередній досвід роботи з різницевою машиною, вирішив збудувати більш складну систему, що ставила перед ним набагато більші технічні та управлінські виклики. Це перевищило його ресурси та можливості на багато порядків. Якби він залишився в рамках більш реалістичної мети, наприклад, завершивши хоча б основну версію різницевої машини, він міг би забезпечити успіх проєкту в рамках доступних можливостей. Натомість він прагнув до створення унікальної та універсальної машини, яка виявилася надмірно складною для свого часу.

З точки зору методологій розробки, Беббідж не застосував інкрементальний або ітеративний підхід. Замість того, щоб реалізувати часткові робочі модулі чи мінімально життєздатний продукт (MVP), він намагався реалізувати ідеально завершену систему одразу, що є помилкою для великих проєктів з високим рівнем невизначеності. Відсутність верифікації кожного етапу, а також недостатній контроль над технічним боргом призвели до того, що в кінцевому підсумку проєкт був зупинений через надмірну складність і технічні проблеми. Беббідж не застосував принципу "fail fast, learn fast", що дозволяє на ранніх етапах виявляти слабкі місця і коригувати стратегію.

У результаті, незважаючи на величезні амбіції і новаторські ідеї Беббіджа, його проєкти не завершилися успіхом через низку проблем в управлінні, недостатнє планування та неправильний підхід до оцінки ризиків. Проте, важливість його ідей була визнана лише через багато років, коли інженери змогли реалізувати його концепції в 1990-х, довівши, що концепції Беббіджа мали потенціал, але потребували більш сучасних умов для успішної реалізації.

Виникнення методологій розробки

Певні методи планування та організації роботи використовувалися ще в античні часи, хоча вони не були систематизованими методологіями, як ми їх розуміємо сьогодні. Єгиптяни, ймовірно, мали свої стратегії та методи управління проєктами для будівництва таких величезних споруд, як Великі піраміди в Гізі.

Будівництво піраміди Хеопса, яке тривало близько 20 років, є прикладом величезного організаційного зусилля, яке вимагало планування, управління ресурсами, координації праці великої кількості робітників та обчислень для досягнення високого рівня точності в архітектурі. Геродот, хоча і не був сучасником цієї події, згадує в своїх творах про тривалість будівництва пірамід, що підтримує уявлення про великий масштаб і складність такого проєкту.

Хоча терміни "методологія розробки" та "управління проєктами" стали популярними лише в останні століття, можна сказати, що єгиптяни вже застосовували принципи, що нагадують сучасні підходи до організації та управління великими проєктами.

Сунь-цзи сказав: “управляти масами — те саме, що управляти небагатьма: річ у частинах і в числі. Вести в бій маси — те саме, що вести в бій небагатьох: річ у формі та назві” (Мистецтво війни, розділ 5).

Ксенофонт у своїх творах, зокрема в "Ойкономікос”, наголошував на необхідності раціонального управління ресурсами, організації праці й досягнення максимального результату з мінімальними витратами. Він розглядав економіку як мистецтво управління домогосподарством, роблячи акцент на ефективності.

В Євангелії від Луки 14:28-32 йдеться про те, що перед початком будь-якої справи слід оцінити свої ресурси. Неважливо, чи плануєш ти будувати, чи вступати у війну, завжди потрібно враховувати витрати. Також відома метафора з Євангелія (Мф. 7:24-27) про будівництво на піску, яка вплинула на казку про трьох поросят.

Адам Сміт, у свою чергу, в "Багатстві народів" (1776), розвинув цю ідею, детально пояснивши вигоди розподілу праці. Він довів, що спеціалізація дозволяє підвищити продуктивність завдяки зосередженню на окремих завданнях, скороченню часу на переключення між операціями та вдосконаленню майстерності.

Ці концепції формують основу економічної науки та раціонального управління працею.

На відміну від формалізованих методологій сучасності, вони не мали детального алгоритму дій або інструментарію для їх реалізації. Це були радше базові принципи, які могли адаптуватися під конкретні обставини.

Наприклад, сучасні методології, такі як Lean Manufacturing чи Agile, містять не лише теоретичні основи, а й конкретні інструменти, які можна використовувати для досягнення ефективності.

"Виробництво — це не купівля за низькими цінами та продаж за високими. Це процес закупівлі матеріалів чесно і, з мінімальними можливими додатковими витратами, перетворення цих матеріалів у товар для споживача. Азарт, спекуляція та гострі угоди лише заважають цьому процесу" (Генрі Форд, "Моє життя і робота").

Генрі Форд зробив революцію в організації праці та промисловому виробництві, впровадивши потокове виробництво та конвеєрну систему (pipeline system). Його внесок базувався на ідеях Адама Сміта про розподіл праці, але він переніс ці принципи на практику в масштабах промислової революції.

Toyota вдосконалила систему Форда, додавши гнучкість. Якщо Форд фокусувався на масовому виробництві одного продукту, Toyota створила систему, яка дозволяла швидко налаштовувати лінії (pipelines) для випуску різних моделей. Вони впровадили підхід "Точно вчасно" (Just-in-Time), який мінімізував запаси, зменшив витрати на зберігання і синхронізував виробництво з попитом. Крім того, принцип Jidoka забезпечував виявлення та усунення дефектів у реальному часі, дозволяючи зупиняти лінію для негайного вирішення проблем.

Ще однією інновацією стало активне залучення працівників до процесу вдосконалення через культуру кайдзен (постійне покращення) і систематичне усунення втрат (Muda). Toyota адаптувала ідеї Форда, зосередившись на створенні цінності на кожному етапі виробництва, що дало змогу випускати продукцію швидше, дешевше і якісніше. Цей підхід забезпечив їм перевагу на ринку, трансформувавши глобальну індустрію.

На початку 20 століття з розвитком
індустріалізації виник підхід Тейлоризму, основою якого була стандартизація, чітке розмежування завдань та зменшення варіативності в процесах. Ця методологія, орієнтована на оптимізацію праці, мала значний вплив не лише на виробництво, а й на сферу програмування.
Фредерік Тейлор визначив мистецтво управління як “точно знати, що ви хочете, щоб люди робили, а потім бачити, що вони роблять це найкращим і найдешевшим способом”.
У 1909 році Тейлор підсумував свої методи підвищення ефективності у своїй книзі “Принципи наукового управління” (Principles of Scientific Management).

У 1960-х роках компанія IBM, один із піонерів у галузі обчислювальної техніки, зіткнулася з серйозними труднощами при розробці складних програмних систем. Ці труднощі отримали назву Second-System Effect – явище, коли системи другого покоління ставали надмірно складними через прагнення розробників врахувати всі можливі функції та виправити недоліки попередніх версій. Як наслідок, такі проєкти були важкими для управління,

потребували значних ресурсів і часто не відповідали очікуванням. Про Second-System Effect писав Фредерік Брукс в “Міфічний людино-місяць” (1975).

Паралельно з цим постала ширша проблема, яка увійшла в історію як Software Crisis ("криза програмного забезпечення"). Цей термін описує труднощі розробки, управління та підтримки великих програмних систем, які виникли через швидкий технологічний прогрес. Технології розвивалися настільки стрімко, що методи й підходи до програмування не встигали за ними. У результаті багато проєктів виходили за рамки бюджету, перевищували строки розробки або не відповідали початковим вимогам, що спонукало до пошуку нових підходів у сфері інженерії програмного забезпечення.
Термін “криза програмного забезпечення” був вперше введений деякими учасниками першої конференції з програмної інженерії НАТО у 1968 році в Гарміші, Німеччина.

У 1960 роки для оцінки проєктів було розроблено метод PERT (Program evaluation and review technique), що орієнтувався на чітке планування і управління ризиками.
PERT пропонує формулу триточкової оцінки часу виконання, однак сама формула базується на емпіричних даних, які зазвичай невідомі заздалегідь. В гнучких методологіях побудованих на емпірицизмі, оцінка відбувається на основі аналізу попереднього досвіду щодо часу виконання певних задач, або подібних до них.

Модель V (V-model) в управлінні проєктами і розробці програмного забезпечення з'явилась в середині 1980-х років. Вона була розроблена як еволюція попередніх моделей розробки, зокрема, Waterfall (каскадної моделі).

Основною ідеєю V-моделі є те, що для кожної стадії розробки програмного забезпечення є відповідна стадія тестування, що забезпечує перевірку на кожному етапі. Таким чином, модель нагадує літеру "V", де одна сторона представляє процес розробки, а інша — процес тестування та верифікації.

Значний прорив у розробці методологій стався з появою Спіральної моделі у 1986 році, запропонованої Баррі Боемом. Спіральна модель (Spiral model) поєднувала переваги класичних водоспадних моделей і прототипування, дозволяючи гнучко адаптувати процес розробки до реальних умов.

У 1990-х роках на тлі швидкого розвитку технологій і вимог до швидкості розробки з'явилася RAD (Rapid Application Development), методологія, яка акцентувала увагу на швидкому створенні прототипів і адаптації до змінюваних вимог. У наступні десятиліття розвинулися інші методології, такі як FDD (Feature-Driven Development), XP (Extreme Programming), що стали популярними в середовищі Agile-розробників.

YAGNI (You Aren't Gonna Need It) — це принцип розробки програмного забезпечення, який наголошує на тому, що не варто реалізовувати функціонал або додавати можливості в систему, доки в них немає прямої необхідності.

Принцип YAGNI був сформульований розробниками методології Extreme Programming (XP) — Кентом Беком (Kent Beck), Роном Джеффрісом (Ron Jeffries) та іншими. Розробники часто схильні додавати функції “про всяк випадок”, вважаючи, що вони можуть стати у пригоді в майбутньому. Однак досвід показує, що більшість таких функцій або не використовуються взагалі, або виявляються непотрібними в контексті реальних вимог. YAGNI закликає відмовитися від передчасної реалізації, залишаючи функціонал, який дійсно необхідний, на майбутнє, коли це стане очевидно.

Agile-методології дозволяють працювати більш гнучко та взаємодіяти з клієнтами через постійну зворотну реакцію.

У цей же період з'являються методи, які підкреслюють важливість організаційної культури, такі як Crystal.

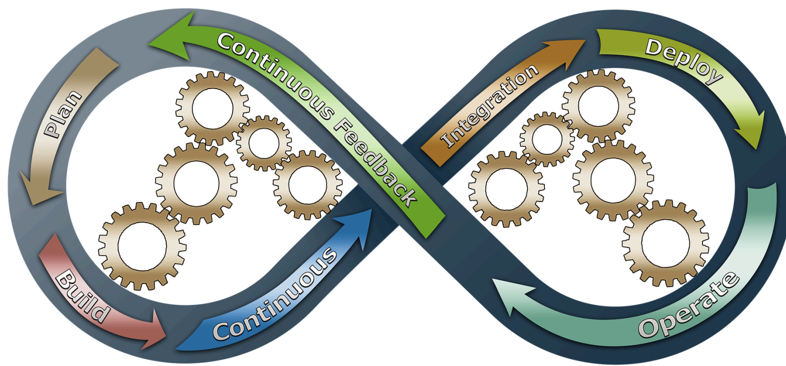
Фреймворк Scrum був чітко й лаконічно описаний в Посібнику зі Скраму (Scrum guide, 2009).

Згодом розвивається DevOps.
В 2009 році для O'Reilly розробники John Allspaw і Paul Hammond зробили доповідь “10+ Deploys Per Day: Dev and Ops Cooperation at Flickr".

Деякі думки з цієї доповіді:

DevOps - це методологія, яка заохочує розробників (dev) думати як сисадміни (ops), а сисадмінів як розробників.
Також була представлена trunk-based model для Git.

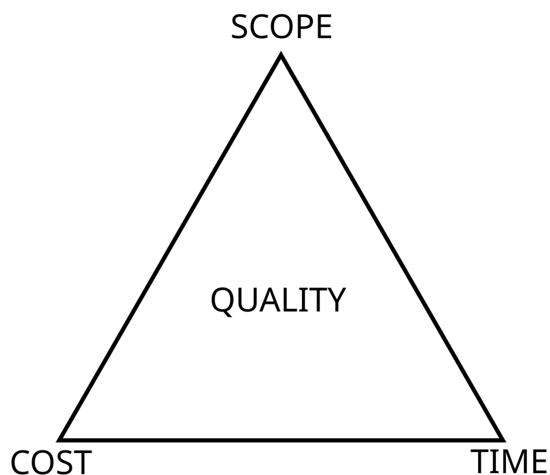
Принципи ДевОпс однозначно базуються на деяких принципах Extreme programming (1999).
Методологія "Twelve-factor app" (2011) від Хероку також вплинула на ДевОпс практики.



Оскільки Clean Agile, Scrum та DevOps активно залучають практики Extreme programming, потрібно зрозуміти чому вжито слово Екстремальне (Extreme).
Екстремальне програмування названо так, тому що воно доводить певні практики до їх максимуму і часто відходить від класичних підходів, які здаються інтуїтивно зрозумілими. Наприклад, Екстремальне програмування наполягає на TDD, воно дивує архітекторів коли ставить BigDesignUpFront проти YagNi, ProgramInTheFutureTense проти DoTheSimplestThingThatCouldPossiblyWork. XP також каже що потрібно мінімізувати час між розробкою, інтеграцією та деплоєм (CI/CD). Реліз в XP може бути десятки разів на день шляхом введення практик continuous integration (CI) та continuous delivery (CD).

Фундаментальні поняття

Нижче описані поняття та принципи, які є фундаментальними для всіх методологій розробки.



Project Management Triangle (трикутник управління проєктами), також відомий як трикутник обмежень, описує взаємозв'язок між трьома основними аспектами управління проєктом: часом, вартістю та якістю (або обсягом). Ці три елементи знаходяться в постійній взаємодії, і будь-яка зміна одного з них може вплинути на інші.

1. Час: Це термін, за який проєкт має бути завершений. Визначає, скільки часу потрібно на виконання проєкту.

2. Вартість: Це бюджет проєкту або кількість ресурсів, що необхідні для його реалізації. Включає як фінансові витрати, так і використання людських ресурсів.

3. Якість: Це кінцевий результат проєкту, його характеристики та вимоги до виконання. Якість визначає, наскільки добре проєкт задовольняє потреби та очікування замовників.

Трикутник показує, що ці три аспекти взаємопов'язані. Наприклад:

- Якщо скоротити час для виконання проєкту, може знадобитися збільшити бюджет або знизити якість.
- Якщо зменшити бюджет, це може потребувати більше часу або вплинути на якість результату.
- Якщо змінити вимоги до якості (обсягу), це може вплинути на час і бюджет проєкту.

Управління цими трьома обмеженнями є основним завданням керівника проєкту, щоб знайти баланс між ними та досягти успішного завершення проєкту в межах заданих обмежень.

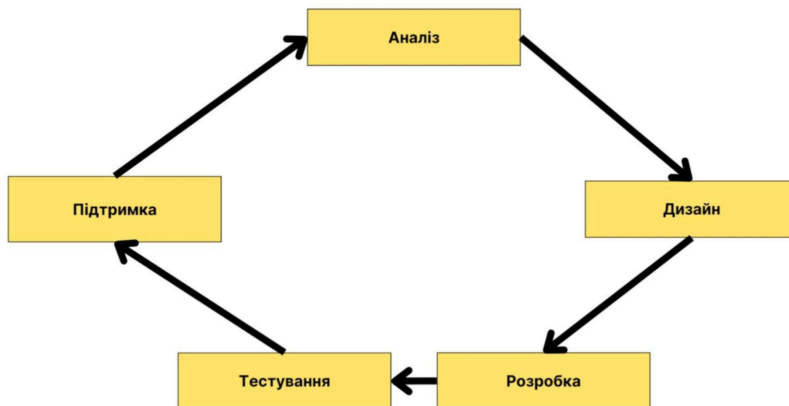
Проєкт — тимчасове зусилля, здійснене для
створення унікального продукту, послуги або результату.
Тимчасовий характер проєктів передбачає початок і кінець роботи над проєктом або його фазою.
Проєкти можуть бути самостійними або частиною портфоліо.

Продукт – це результат творчої та інженерної діяльності, створений для задоволення потреб користувачів чи вирішення проблем.

(Продукт може бути послугою, фізичним продуктом або чимось більш абстрактним.   Ціль Продукту (Product Goal) — це довгострокова мета для Скрам Команди. Вони повинні виконати одну ціль (або відмовитись від неї), перш ніж братись за наступну.)

Цикл розробки продукту починається з ідеї, яка проходить стадії дослідження ринку, створення вимог, проєктування, розробки, тестування, запуску, підтримки та подальшого вдосконалення. На кожному етапі важливо дотримуватися принципів адаптивного

планування та залучати всіх зацікавлених осіб до процесу. Успішний цикл розробки забезпечує не лише якість продукту, а й його відповідність ринковим умовам.



Закон Конвея стверджує, що організаційна структура команди впливає на архітектуру продукту. Наприклад, якщо команда розділена на кілька підгруп, кожна з яких займається окремою частиною системи, продукт, швидше за все, буде складатися з ізольованих модулів. Відповідно відкрита комунікація покращує якість продукту.

Закон Брукса наголошує, що додавання нових розробників до проєкту на пізніх етапах часто лише уповільнює процес через збільшення часу на комунікацію та навчання. Ці закони допомагають зрозуміти взаємозв'язок між організацією команди та ефективністю розробки.

Фред Брукс дає формулу збільшення часу на комунікацію відносно збільшення кількості учасників команди.

Кількість можливих зв'язків між учасниками визначається за формулою:
$C = n(n-1)/2$,

де:
C — кількість комунікаційних зв'язків,

n — кількість учасників команди.

Bus factor визначає, скільки ключових членів команди може залишити проєкт без шкоди для його розвитку. Чим вищий цей показник, тим стійкіший проєкт.

Second-system effect описує ситуацію, коли друга версія продукту стає надто складною через прагнення включити всі попередні ідеї та нові функції. Це може призвести до збільшення витрат, часу розробки та навіть провалу продукту.

Колективний досвід — це сумарний обсяг знань, навичок, якими володіє команда. Він охоплює внесок кожної окремої людини, незалежно від того, чи працювали вони разом у команді. Такий досвід формується на основі індивідуальної експертизи та спільного обміну інформацією в межах колективу.

Командний досвід — це знання та навички, які команда здобуває разом у процесі спільної роботи. Він формується через взаємодію, координацію дій та вирішення завдань у рамках конкретного проєкту чи діяльності. Це досвід, який стосується саме роботи команди як єдиного цілого.

Розділення на колективний і командний досвід важливе, оскільки вони по-різному впливають на прийняття рішень та організацію роботи. Командний досвід є спільним, тобто всі учасники команди розуміють і погоджуються з ним, що забезпечує основу для ухвалення зважених і ефективних рішень. Це робить його критично важливим для узгодженої роботи в межах методологій розробки, таких як Scrum або XP, де накопичення і аналіз цього досвіду є ключем до постійного вдосконалення.

Колективний досвід, хоча й охоплює ширший спектр знань, не завжди може бути повністю узгодженим, оскільки включає індивідуальні підходи різних учасників. Він важливий для розвитку загальної експертизи та передачі знань.

Common-Knowledge Effect (ефект загального знання) — це феномен групової динаміки, коли учасники дискусії схильні обговорювати лише ту інформацію, яка відома всім, ігноруючи унікальні знання окремих членів групи. Це може призводити до менш ефективного ухвалення рішень, оскільки важлива, але нерозкрита інформація не враховується.

Ефект загального знання виникає через комфортність обговорення знайомих тем і уникання конфліктів, пов'язаних з унікальною інформацією. Подолати це можна за допомогою створення атмосфери відкритості, де заохочуються нові ідеї та індивідуальний внесок кожного учасника.

Групи повинні краще приймати рішення, ніж окремі люди, оскільки вони поєднують багато точок зору. Але на практиці група приймає рішення не на інформації, специфічній для кожного члена, а лише на інформації, спільній для всіх. Це ставить під сумнів думку про те, що “дві голови краще, ніж одна”, і допомагає пояснити, чому, незважаючи на поширену мудрість, різноманітність зазвичай не робить команди кращими.

Confirmation Bias (упередженість підтвердження) — це когнітивне спотворення, коли людина схильна звертати увагу лише на ту інформацію, яка підтверджує її попередні

переконання, ігноруючи факти, що їм суперечать. Це спотворення може впливати на прийняття рішень, змушуючи людину відкидати об'єктивні докази, якщо вони не узгоджуються з її поглядами.

Атрибутивне упередження (Attribution bias) — це когнітивна тенденція пояснювати поведінку людини через її характеристики або внутрішні риси, а не через зовнішні ситуаційні фактори. Це може призводити до неправильних або упереджених суджень про інших, оскільки не враховуються обставини або зовнішні чинники, що можуть впливати на їхні дії. Наприклад, якщо хтось запізнився на зустріч, ми можемо приписати це його недбалості або неорганізованості (внутрішні фактори), замість того, щоб взяти до уваги зовнішні обставини, такі як затори чи непередбачена ситуація.

Атрибутивне упередження може впливати на різні аспекти соціальних взаємодій, зокрема на те, як ми оцінюємо поведінку інших, і може сприяти формуванню стереотипів або непорозумінь.

Ітеративні гнучкі методології розробки з їх циклом аналізу прогресу та проблем розроблені для мінімізації різних типів упереджень (confirmation and attribution bias) методом спільного відкритого аналізу та обговорення. Відкритість зменшує недовіру, яка є однією з головних проблем у команді. Недовіра виникає через неповагу, упередження, відсутність об'єктивних оцінок та аналізу.

Пасивна агресія в команді проявляється, коли члени колективу не виражають свої проблеми або незадоволення відкрито, а використовують непрямі методи, щоб передати свої емоції або послання. Це може створювати напругу, знижувати ефективність роботи та впливати на атмосферу в команді.

Для уникнення цього були описані цінності Scrum та Extreme programming (XP).

Серед основних цінностей XP — це комунікація, зворотний зв'язок, простота, сміливість і повага. **Комунікація** підкреслює важливість тісної взаємодії між розробниками та замовниками, зворотний зв'язок забезпечує постійну адаптацію до змін вимог, а **простота** (KISS + YAGNI) орієнтується на мінімізацію складності коду для полегшення його підтримки. **Сміливість** дозволяє приймати ризиковані, але корисні рішення, а **повага** допомагає підтримувати здорові відносини в команді та з замовниками. Ці цінності сприяють створенню більш адаптивного та стійкого до змін процесу розробки.

Скрам додає ще Focus (сфокусованість) як цінність. Це для того, щоб розробники зосереджувалися на цілі ітерації (спринта) і не робили “дурну роботу”.

Типи організації команди

Опис трьох типів організації команди:

1. Мурашина модель (плоска модель без тренера): У цій моделі команда працює як єдине ціле, де всі члени взаємодіють між собою та сприяють досягненню спільної мети. Команда функціонує, подібно до мурашиної колонії, без чітко визначеного лідера. Натомість, є чіткі алгоритми та правила, що допомагають організувати роботу і забезпечити ефективність взаємодії між учасниками. Кожен має чітке розуміння своїх дій у процесі, що дозволяє команді працювати злагоджено навіть без лідера.

2. Плоска модель з тренером: У цьому підході команда має рівний рівень ієрархії, де немає жорсткої вертикалі. Кожен має можливість висловлювати свої ідеї та брати на себе відповідальність за проєкт. Тренер або ментор допомагає команді, надаючи підтримку та направляючи її до розвитку, але не виконує функції керівника, а виступає більше як консультант або фасилітатор.

Роберт Мартін в книзі "Чистий Agile" пише: "Чи потрібен тренер Agile-команді? Коротка відповідь: «Ні». Довша відповідь: «Іноді»". Роберт Мартін каже що Agile-тренер має навчити команду основним принципам за декілька тижнів, а далі він не потрібен. Він також каже, що при цьому роль Agile-коуча потрібна, але вона може бути призначена певному розробнику, умовному лідеру команди (team leader), або тому хто в команді за старшого в цей день. Agile-коуч слідкує, щоб принципи Agile виконувалися. (В такому випадку, плоска команда немає лідера, але може мати Agile-коуча, який також може бути розробником. Це гнучка версія лідера команди, якщо дозволите.)

3. Ієрархічна модель: Це традиційний підхід, де команда організована за чіткою ієрархією, з вищими рівнями керівництва та визначеними підлеглими. Лідери відповідають за прийняття рішень, а решта команди виконує покладені завдання. Цей тип організації може бути ефективним у великих або складних проєктах, де потрібна чітка структура та контроль. Має свої ризики, а саме схильна до антипатерна Диктатор, та часто має низький bus factor.

Ці моделі можуть бути адаптовані або змішуватись залежно від потреб конкретної команди та проєкту.

Типи методологій розробки

Класифікація методологій розробки:

1. Гнучкі методології та жорсткі (контрактні).

2. Ітераційні та проєктні методології.

3. Методології з кросфункціональними та функціональними командами.

4. Методології з плоскими та ієрархічними командами.

Ось опис різних типів методологій розробки:

Гнучкі методології та жорсткі (контрактні):

Гнучкі методології (Agile): Ці методології базуються на Agile маніфесті. Вони зосереджуються на постійному зворотному зв'язку з клієнтом і забезпечують інтеграцію нових вимог у процес розробки. Agile фреймворки та методи включають Scrum, Kanban, XP (Extreme Programming), і т.д.

Маніфест Agile описує основні принципи гнучких методологій розробки:

- Люди та співпраця важливіші за процеси та інструменти.
- Працюючий продукт важливіший за вичерпну документацію.
- Співпраця із замовником важливіша за обговорення умов контракту.
- Готовність до змін важливіша за дотримання плану.

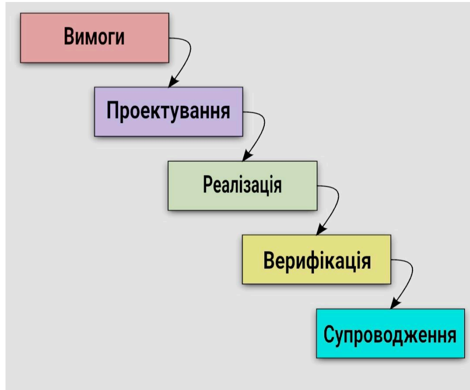
Хоча, цінності, що справа важливі, ми все ж цінуємо більше те, що зліва.

Жорсткі методології (контрактні методології): Вони орієнтовані на чітке визначення вимог і специфікацій на початку проєкту, з формалізованими угодами і контрактами між замовником та розробниками. Такі методології часто використовуються в класичних водоспадних підходах (Waterfall), де весь проєкт спочатку планується, а потім реалізується відповідно до цих планів.

Ітераційні та проєктні методології:

Ітераційні методології: Ці методології передбачають розробку продукту через серію ітерацій або циклів, кожен з яких має своє завершення та реліз. Кожна ітерація дозволяє перевірити поточний прогрес, коригувати помилки та адаптувати продукт відповідно до нових вимог. Scrum і Kanban є прикладами таких підходів.

Проєктні методології: Вони фокусуються на плануванні та виконанні всього проєкту як одного цілого. Підхід часто передбачає чітко визначену структуру, де кожен етап проєкту має чітке завдання і кінцеву мету. Waterfall є типовим прикладом проєктної методології.



(Waterfall model, Wikipedia, author Bunyk, License CC BY 3.0).

Поширеною помилкою є, коли Agile/XP/Scrum команди починають працювати як міні-Waterfall, наприклад, працюючи в ізоляції або надмірно документуючи user stories, що порушує ітеративний процес і затримує тестування.

У реальному Agile процесі всі учасники проєкту мають тісно співпрацювати на кожному етапі. Якщо команда починає працювати ізольовано, наприклад, розробники зосереджуються тільки на написанні коду, а тестувальники чекають на готовий продукт для тестування, то це створює ефект "мікроводоспаду". Тестування розпочинається тільки після завершення розробки, що призводить до затримок і на стадії тестування, і в загальному циклі роботи.

Методології з кросфункціональними командами:

У таких методологіях команда складається з учасників, які мають різні навички та компетенції, що дозволяє їм виконувати різні аспекти проєкту. Кожен член команди може працювати на кількох етапах розробки, від проєктування до тестування, що підвищує ефективність співпраці та прискорює процес розробки.

Agile (особливо Scrum) є прикладом методології, яка активно використовує кросфункціональні команди. У Scrum, наприклад, команди складаються з розробників, тестувальників, аналітиків, дизайнерів тощо, і всі працюють разом над виконанням завдань в межах спринтів. Цей підхід дозволяє швидко адаптуватися до змін, оскільки всі ролі перебувають в одній команді і можуть оперативно взаємодіяти.

Методології з функціональними командами:

У цих методологіях команда складається з учасників, які мають однакові функціональні спеціалізації, наприклад, команда тільки для розробників, команда для тестування,

команда для аналітиків тощо. Кожна команда зосереджена на виконанні своєї конкретної функції, і співпраця між командами часто організована через чітке планування та комунікацію.

Waterfall або класичні проєктні методології часто використовують функціональні команди, де процес розробки чітко розподілений між спеціалізованими групами. Наприклад, розробники можуть повністю завершити свою частину роботи, перед тим як тестувальники почнуть свою, що може призводити до більш тривалих циклів зворотного зв'язку.

Керування ризиками

Ітерації в методології розробки програмного забезпечення є важливим елементом процесу, який забезпечує поділ складного проєкту на менші цикли. Це дозволяє командам досягати поступових результатів, коригувати хід розробки на основі реальних даних і знижувати загальні ризики. Розглянемо основні переваги ітераційного підходу:

Зворотний зв'язок.
Кожна ітерація завершується демонстрацією функціоналу, що дозволяє отримати відгуки від зацікавлених осіб, таких як клієнти або користувачі. Це дає можливість на ранніх етапах виявити помилки або неточності в реалізації, а також уточнити вимоги, що допомагає зберегти якість і відповідність продукту очікуванням. Постійний процес зворотного зв'язку знижує ймовірність суттєвих помилок на наступних етапах розробки.

Адаптація оцінок.
Під час кожної ітерації команда отримує актуальні дані про реальні витрати часу та ресурсів. Це дає змогу коригувати попередні оцінки та робити прогнози на основі фактичного досвіду, що зменшує невизначеність і підвищує точність планування на майбутні етапи.

Управління складністю.
Поділ проєкту на ітерації дозволяє значно знизити когнітивне навантаження на команду, адже розробка здійснюється через поступове вирішення менших завдань. Це дозволяє зосередитися на найбільш важливих частинах продукту, зменшуючи складність і полегшуючи управління проєктом. Таким чином, ітерації дозволяють ефективніше вирішувати проблеми і контролювати процес на кожному етапі.

Постійне вдосконалення.
Кожна ітерація супроводжується аналізом результатів і ретроспективою, що дозволяє команді ідентифікувати проблеми в процесах і вдосконалювати їх. Це стимулює оптимізацію робочих процесів, підвищує продуктивність і дозволяє швидше реагувати на зміни в вимогах або умовах, зберігаючи стабільність і ефективність роботи.

Загалом, ітераційний підхід дозволяє забезпечити високу гнучкість в управлінні проєктами, знижує ризики, покращує якість продукту і сприяє постійному вдосконаленню процесу розробки.

Ітерація — це циклічний процес в методологіях розробки, під час якого команда виконує певні завдання, такі як планування, розробка, тестування та оцінка результатів. Ітерація має визначену тривалість (зазвичай 1-4 тижні) і забезпечує постійне вдосконалення процесу розробки, а також отримання зворотного зв'язку для подальших коригувань.

Інкремент — це частина функціональності або нові можливості, що додаються до продукту після кожної ітерації. Інкремент є завершеною, самодостатньою частиною продукту, яка може бути використана або продемонстрована клієнтам, і в кінцевому підсумку складає основну частину фінального продукту. Інкремент є результатом роботи команди, який розвивається з кожною ітерацією.

Для зменшення ризиків важливе вчасне і незалежне тестування.

Незалежне тестування, описане Гленфордом Маєрсом у "The Art of Software Testing", полягає в перевірці програмного забезпечення командою або фахівцями, які не брали участі в його розробці, що забезпечує об'єктивність і зменшує ризик упередженості. Маєрс наголошує, що незалежність може бути внутрішньою (окрема команда) або зовнішньою (наймання експертів), але важливо зберігати співпрацю між тестувальниками і розробниками. Такий підхід сприяє виявленню прихованих помилок, покращенню якості продукту та є критично важливим для систем із високими вимогами до надійності. Якщо тестування проводить інший програміст/тестувальник із тієї ж команди, це також може забезпечити певний рівень незалежності, оскільки він не брав безпосередньої участі в розробці конкретного функціоналу. Такий підхід дозволяє уникнути упередженості автора коду, але Маєрс зазначає, що для досягнення максимальної об'єктивності краще, щоб тестування виконувала окрема команда або зовнішні фахівці.
У гнучких кросфункціональних командах незалежність тестування досягається шляхом розподілу відповідальності: тестування виконує інший член команди, який не розробляв перевіряємий функціонал. Такий підхід забезпечує швидкий зворотний зв'язок і зберігає принцип незалежності в межах команди. Важливо, щоб тестувальник володів знаннями про продукт і підтримував ефективну комунікацію з розробником, уникаючи конфлікту інтересів, але зберігаючи об'єктивність.

Оцінка задач

Коли ми вводимо поняття "людино-година", ми робимо припущення: що всі люди (зокрема, інженери ПЗ) мають рівні вміння, а отже можна чітко визначити час на виконання певної задачі. Це припущення вже невірне, коли в команді є старші та молодші спеціалісти, які роблять одне й те саме за різний час.

Крім того, людино-година рахується окремо для кожного працівника, що не вірно для командної роботи, де доречніше було б рахувати в “командних-годинах”.
Щоб вирішити ці недоліки поняття "людино-година", були створені story points.
Під людино-годиною варто розуміти “ідеальні години”, які не враховують форс-мажорні перепони.

Story point (Очки складності завдання) - це одиниця виміру, яка використовується в методології розробки програмного забезпечення для оцінки складності задачі. Вона використовується для приблизної оцінки часу і зусиль, не прив'язуючись до конкретної часової рамки.
Коли команда розробників оцінює задачу, вони приймають у розгляд три основні фактори: складність, обсяг роботи та ризики. Вони потім призначають задачі певну кількість story points, яка відображає загальну складність. Чим більше story points, тим складніше завдання. Це дозволяє команді спрогнозувати, скільки робочих годин знадобиться на виконання задачі і планувати роботу відповідно.
Між story point та часом немає прямопропорційної залежності.

Аксіоми сторіпойнтів:

- Найменший сторіпойнт 1.
- Сторіпойнти лінійні, тобто 4 сторіпойнти = 2 + 2 сторіпойнти.
- Не існує найбільшого сторіпойнта.

Сторіпойнт вказує на конкретну реферну задачу, яку кожен програміст може виконувати за різний час, залежно від умінь.

Команда може домовитися, що коли задача більше 10, або 20 сторіпойнтів, її розбивають на підзадачі. Декомпозиція задачі на підзадачі дозволяє раніше починати тестування.

Velocity (швидкість) у story points потрібен для ефективного прогнозування та планування роботи. Він дозволяє команді визначити, скільки завдань вона може виконати за спринт, і дає можливість стейкхолдерам побачити реалістичні терміни завершення проєкту. Це зменшує ризики перевантаження команди, допомагає уникнути завищених очікувань і створює прозорість у процесах.
Без velocity складно оцінити динаміку продуктивності та стабільність роботи команди.

Ємність (Capacity) — це максимальна кількість роботи, яку команда може взяти на себе в наступному спринті. Команда може швидко встановити ємність, виходячи з середньої швидкості (velocity), а потім коригувати її враховуючи доступність кожного під час наступного спринту.
Навантаження (Load) — це кількість роботи, обрана командою для поточного спринту. Це кількість роботи, яку команда планує завершити протягом спринту. Воно повинно бути менше або рівним ємності.

Швидкість (Velocity) — це кількість роботи, завершена в попередніх спринтах. Це міра минулої продуктивності команди. Зазвичай дивляться на останні три-п'ять спринтів і беруть середнє значення їхньої швидкості.
Ось формули:
швидкість >= ємність >= навантаження, буфер = ємність - навантаження.
Різниця між ємністю та навантаженням — це ваш планувальний буфер.

Ви не зможете визначити адекватну швидкість команди, якщо команда постійно працює в стресовому режимі.

Sustainable pace (сталий темп) — це принцип в Agile-методологіях, який вказує на необхідність підтримки робочого темпу, який можна безпечно підтримувати протягом тривалого часу, не втомлюючи команду та не знижуючи продуктивність. Цей принцип підкреслює важливість збалансованого навантаження, яке дозволяє командам ефективно працювати, не вигораючи, і зберігати високу мотивацію на довгостроковій основі.

Знайдіть ідеальну швидкість вашої команди, яка залишатиметься незмінною протягом усього проєкту. Кожна команда різна. Вимагання від цієї команди збільшити швидкість, щоб відповідати цій команді, фактично знизить її швидкість у довгостроковій перспективі. Тож якою б не була швидкість вашої команди, просто прийміть її, бережіть її та використовуйте, щоб будувати реалістичні плани.

Забудьте тимчасового про естімацію в годинах. Скільки сторіпойнтів (story points) в спринті ви можете зробити - головне питання. Час фігурує тільки в розмірі спринта, все. Вам потрібно визначити скільки сторіпойнтів ви можете видати за спринт (наприклад 2 тижні, враховуючи всі мітинги).

Сторіпойнт відносна одиниця. Наприклад, якась задача 2 сторіпойнти, а інша в два рази більша, або в два рази складніша, буде 4 сторіпойнти. Все. Чітко встановіть цю референу задачу.
Оцінку можна робити в числах Фібоначчі, або просто 1,2,3,4,5,... Немає верхньої межі (на практиці буде десь 21).

Покер планування - це метод оцінки задачі.

Ключові особливості покер планування:
1. Колективна оцінка.
2. Анонімна оцінка, до моменту оголошення результатів.

Покер планування з'явилося на початку 2000-х років і було популяризоване у впливовій книзі "Agile Estimating and Planning" (2005) by Mike Cohn. Там же були описані story points.

До речі цю книгу в українському перекладі видало видавництво Фабула. ("Оцінювання і планування в Agile", Майк Кон, 2019).



На покер плануванні команда розробників, в їх числі тестувальники, оцінюють юзер сторі (задачу). Перед цим людина, яка найкраще може описати задачу пояснює її всім учасникам.
Оцінюють всі компетентні члени команди. Практика оцінювати окремо для тестування та окремо для розробки має певні проблеми.

Щоб не впливати не рішення один одного й не давити авторитетом, оцінювання проводиться анонімно. Потім всі разом "відкривають карти", тобто оголошують свої оцінки. Якщо оцінки розходяться, відбувається дискусія, щоб побачити, що не було враховано. Зазвичай результат оцінки має бути одностайним, як в суді присяжних.

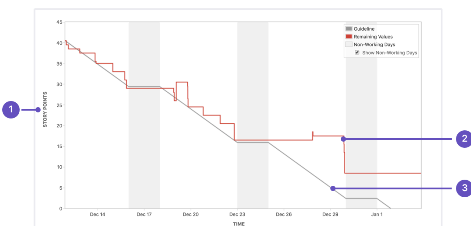
Покер планування покращує довіру в команді та шанс, що задача буде коректно оцінена.
Покер планування зазвичай робиться під час планування спринту, коли всі блокери вирішені.
Під час беклог рефайнменту (грумінгу) також проводять покер планування, щоб побачити чи задача вміститься в спринт.
Якщо є нюанс, який блокує оцінку, тобто щось що не відоме розробникам, тоді проводиться Spike, міні ресерч, або побудова прототипу, щоб дати коректну оцінку.

Оцінку варто робити у сторі пойнтах, які так чи інакше відповідають "ідеальним годинам". Оцінка в ідеальних годинах означає, що це кількість годин, які потрібно на виконання задачі, якщо всі передумови будуть виконані (не вимкнуть світло тощо). Тобто якщо задача оцінена в 5 ідеальних годин, то це не означає що ви її завершите через 5 годин, це означає що конкретно ви витратите на неї 5 годин.

Не існує найбільшого сторі пойнту як й не існує найбільшого числа.
1 сторі пойнт для одного програміста може зайняти 2 години, а для іншого 6 годин, залежно від рівня програміста. Можна уявити, що сторі пойнти це валюта, й кожен програміст живе в різних країнах, де обмін її виконується по іншому курсу (тобто в години). Однак, повинна бути одна реферна задача, яку всі члени команди, розробники та тестувальники, оцінюють однаково в сторі пойнтах. Всі інші задачі відносно цієї реферної. Задача вдвічі більша, або вдвічі складніша, буде мати вдвічі більшу оцінку.

Atlassian рекомендує використовувати story points для Scrum та burndown chart.

Діаграма згоряння (burndown chart) показує обсяг роботи, виконаної в епіку або спринті, і загальну роботу, що залишилася. Діаграми згоряння використовуються для прогнозування ймовірності завершення вашою командою роботи за відведений час. Вони також чудові для того, щоб тримати команду в курсі будь-яких “несправностей прицілу”.



Burndown Chart для двох тижневого спринта виглядає як дві (сірі) похилі лінії з горизонтальною лінією між ними. Горизонтальна лінія позначає вихідні дні, коли ви не працюєте.
Червона лінія спочатку позначає роботу, яку ви повинні виконати за спринт. Коли ви закриваєте таски ця лінія стає ламаною і, в ідеалі, в кінці спринту повинна бути апроксимацією сірої реферної лінії.

Якщо червона лінія нижче сірої, ви працюєте швидше, якщо червона лінія значно вище, ви працюєте повільніше. В ідеалі, ви маєте працювати лінійно, тобто близько до еталонної (реферної) лінії на burndown chart. Щоб це відбулося, естімація повинна бути релевантною.

Якщо ви використовуєте story points, тоді вам не потрібно ще додатково робити оцінку в годинах.
Atlassian рекомендую не брати задачу, яка буде виконуватися більше 16 годин (або певну кількість сторі пойнтів), тобто потрібно ділити задачі ретельніше.

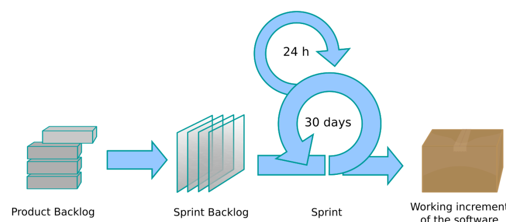
Якщо ви кросфункціональна команда, але на покер-плануванні тестувальник оцінює задачу окремо від розробників та дизайнерів графічного інтерфейсу, це свідчить про певну непослідовність і протиріччя в підході. Якщо ви одна команда, то задачу мають оцінювати

всі учасники. Як це зробити, якщо не всі є фулстеками (універсалами)? Фулстеками бути не обов'язково, але, наприклад, бекенд-розробник повинен вміти оцінити дизайн, хоча б приблизно. Для цього йому слід розуміти, як цей дизайн реалізується. Базова задача на 1 сторі пойнт має бути зрозумілою всім.

Можна по іншому зробити, а саме: мати по два спеціалісти на кожну частину роботи. І нехай вони оцінюють, це калібрує оцінки, але все ж одиниця складності має бути однакова для всіх. Тобто, якщо задача на тестування в 1 сторі пойнт та задача на графічний дизайн в 1 сторі пойнт, то їхня складність однакова.

Scrum

Scrum описаний в Scrum Guide (14 pages).



(Scrum sprint, author Lakeworks, License CC BY-SA 4.0)

Скрам був розроблений насамперед як фреймворк для розробки програмного забезпечення. Він враховував попередні напрацювання інженерів в таких методологіях як FDD та XP. Можна сказати, що шлях до Скраму та Extreme Programming (XP) почався в 1960-х роках, коли виник “Software crisis” та стихійна розробка з мікроменеджментом. Кен Швабер та Джефф Сазерленд вперше спільно представили Scrum на конференції OOPSLA у 1995 році.

Scrum – це фреймворк для гнучкого управління проєктами, орієнтований на ітеративну розробку продуктів. Він акцентує увагу на цінності команди, взаємодію зі стейкхолдерами та адаптацію до змін. Робота в Scrum організована у короткі цикли, названі спринтами, які зазвичай тривають 1-4 тижні. У кінці кожного спринту команда презентує потенційно готовий до використання продукт.

Скрам події (церемонії)

1. Планування спринту (Sprint Planning) – визначення цілей спринту та формування беклогу спринту.
2. Щоденні мітинги (Daily Scrum) – короткі зустрічі (15 хвилин) для синхронізації команди.

3. Огляд спринту (Sprint Review) – демонстрація виконаної роботи зацікавленим сторонам (стейкхолдерам).
4. Ретроспектива (Sprint Retrospective) – аналіз процесів команди для їхнього вдосконалення.
5. Беклог рефайнмент (грумінг) - уточнення вимог, підготовка до Планування спринту.

Суворо визначена тривалість спринта необхідна для того, щоб чітко бачити результати роботи після кожного спринту та зосередитися (фокусуватися) на конкретних завданнях, уникаючи зайвих відхилень.

Спринт зупиняється тільки коли його ціль неактуальна, а не тоді коли ви не встигаєте. Наступний спринт починається одразу після попереднього, без прогалин.

Дейлі мітинг (стендап) повинен бути не більше 15 хвилин. В ньому беруть участь "Розробники" - всі, хто робить щось для інкременту в спринті. Інші гості можуть бути хіба як спостерігачі, але на них час не виділяється.
Кожен учасник Дейлі мітингу каже, як його успіхи, чи все йде по плану. І коротко каже, які в нього виникли проблеми, щоб потім йому хтось допоміг, після мітингу. Дейлі мітинг (Дейлі Скрам) принципово не є мітингом для звіту, звітування, це мітинг для синхронізації.

Часові рамки та порядок беклог рефаймента не прописані, вони визначаються по необхідності.

У Scrum встановлені таймбокси для подій: Sprint Planning – до 8 годин для місячного спринту (пропорційно коротший для менших спринтів), Daily Scrum – 15 хвилин, Sprint Review – до 4 годин для місячного спринту, Sprint Retrospective – до 3 годин для місячного спринту. Таймбокси допомагають команді ефективно планувати, синхронізувати роботу, обговорювати прогрес і покращення продукту.

Покращення беклогу (Backlog Refinement) не є подією з фіксованим часовим обмеженням, але це не означає, що воно не може бути таким. Команда просто повинна виділити достатньо часу на обробку беклогу протягом спринту, щоб беклог був впорядкованим і організованим до наступного планування спринту.

Власник Продукту (Product Owner) відповідає за максимізацію цінності продукту, створеного Скрам Командою, та ефективне управління Беклогом Продукту, включаючи визначення та комунікацію Цілей Продукту, створення й упорядкування елементів Беклогу, а також забезпечення його прозорості та доступності. Хоча Власник Продукту може делегувати ці завдання, він залишається відповідальним за їх виконання. Його рішення, які впливають на зміст і пріоритетність Беклогу, повинні поважатися всіма членами організації. Власник Продукту є однією особою, яка представляє інтереси зацікавлених сторін і приймає остаточні рішення щодо пріоритетів у Беклозі.

5 цінностей Scrum

1. Сміливість (Courage) – мати відвагу брати відповідальність та працювати з викликами.
2. Зосередженість (Focus) – концентруватися на цілях спринту.
3. Почуття обов'язку (Commitment) – бути залученим до роботи та підтримувати команду.
4. Повага (Respect) – цінувати вміння та внесок кожного члена команди.
5. Відкритість (Openness) – бути відкритими до нових ідей та змін.

3 опори (стовпи) Scrum

1. Прозорість (Transparency) – всі процеси, результати та прогрес мають бути зрозумілими.
2. Перевірка (Inspection) – постійне оцінювання роботи для виявлення відхилень.
3. Адаптація (Adaptation) – швидке реагування на зміни або проблеми.

Ролі в Скрам команді

1. Scrum Master – фасилітатор процесів, допомагає команді дотримуватись принципів Scrum.
2. Product Owner – відповідає за формування бачення продукту та пріоритизацію беклогу. Має остаточне слово щодо задач в беклозі та їх пріоритету.
3. Development Team – виконує роботу з розробки та доставки продукту.

Скрам не вимагає T-shaped спеціалістів, хоча це буде перевагою.
T-shaped спеціалісти – це члени команди, які мають глибокі знання в одній області та базові навички в інших.
Кросфункціональна команда – група людей із різними компетенціями, здатна повністю виконувати поставлені задачі без зовнішньої допомоги.

Definition of Ready та Definition of Done

Definition of Ready (DoR) – критерії, що визначають, коли задача готова до Sprint planning (наприклад, чітке формулювання вимог). Прямо не фігурує в Scrum guide, але по суті випливає з нього.
Дивіться критерії INVEST для user story.

Definition of Done (DoD) – критерії, що вказують на завершеність задачі (тестування, документація, готовність до релізу).
До речі Екстремальне програмування наполягає на автоматизації приймальних тестів (Acceptance tests) що засвідчують Definition of Done.

Практика приймальних тестів (зазвичай функціональні тести) в основному є досить прямолінійною. Бізнес (зазвичай через бізнес-аналітиків) пише формальні тести, які описують очікувану поведінку кожної користувацької історії (user stories), а потім розробники автоматизують їх виконання.

Ці тести готуються на етапі підготовки, до початку ітерації, коли ще не було розроблено код для історій, що перевіряються. Розробники включають ці автоматизовані тести до процесу безперервної інтеграції (CI/CD pipeline), що дозволяє постійно перевіряти відповідність коду вимогам. Тести стають частиною визначення готовності (Definition of Done) для кожної

історії. Тобто історія не вважається специфікованою, поки не буде написаний її тест прийняття, і вона не вважається завершеною, поки цей тест не пройде успішно.

Перед покер плануванням юзер сторі (user story) повинна відповідати Definition of Ready, тобто бути придатною до виконання за один спринт та до планування спринту. Definition of Ready для конкретної юзер сторі визначається на зустрічі під назвою Backlog refinement. Ця зустріч не має жорстко встановлених термінів та графіку, й може відбуватися під час спринта із залученням тільки деяких членів команди, зокрема, виконавців. В Backlog refinement так чи інакше залучені Продукт овнер (та Бізнес-аналітик) і, можливо, лідер команди (team leader). Під час рефайнменту даються загальні оцінки, щоб можна було визначити чи задача вміститься в спринт, з урахуванням team velocity. Якщо розробники не можуть оцінити задачу, береться час на дослідження (Spike).
Спайк (Spike) — це експеримент, обмежений часом, який дає змогу розробникам програмного забезпечення визначати та оцінювати юзер-сторі.
Під час скелелазіння іноді ви не можете піднятися далі, тому що нема за що вхопитися. Щоб піднятися далі, потрібно вбити у поверхню скелі спайк (шип). Спайк не наближає вас до вершини, а прокладає шлях, по якому ви можете продовжувати сходження.

Беклог спринту складається з мети спринту (чому), набору елементів продуктового беклогу, вибраних для спринту (що), а також з плану дій для створення інкременту (як).

Ця структура допомагає зберігати ясність у команді, фокусуючи увагу на ключових аспектах проєкту.

Під час спринту:
- Не вносяться зміни, які б могли загрожувати меті спринту;
- Якість не знижується;
- Беклог (backlog) продукту уточнюється за потреби; і, -Обсяг (scope) може бути уточнений та переглянутий з Власником продукту, коли стане відомо більше.
Спринт можна скасувати, якщо ціль спринту застаріє. Лише Власник продукту має право скасувати спринт.

Scrum вимагає від Скрам-майстра створення середовища, де:
1. Власник продукту розміщує роботу для складної проблеми у Backlog продукту.
2. Команда Scrum перетворює відібрану роботу на приріст вартості (Інкремент цінності) під час спринту.
3. Команда Scrum та її зацікавлені сторони (stakeholders) оглядають результати та вносять корективи на наступний спринт.
4. Процес повторюється.

Міфи про Скрам

- В Скрам команді всі повинні бути Сеньйорами, досвідченими програмістами.

- В Скрам команді всі повинні бути універсалами, фулстеками.
- Якщо ви не встигаєте зробити все заплановане на спринті, ви його відміняєте.
- Дейлі міт потрібен для звітування перед начальством.
- Планування та оцінка задач відбувається тільки на плануванні спринту.
- Скрам не підходить для великих та критично важливих проєктів.

Nexus

Nexus Framework – це фреймворк, орієнтований на координацію кількох (3-9) Scrum-команд, які працюють над одним продуктом. Його створено, щоб подолати труднощі масштабування Scrum у великих проєктах, зокрема зростання складності інтеграції. Nexus додає до стандартного Scrum елементи, такі як Nexus Integration Team (команда інтеграції), яка відповідає за забезпечення безперервної інтеграції роботи всіх команд. У фреймворку зберігається звична структура подій (Daily Scrum, Sprint Review тощо), але додаються загальні для представників всіх команд зустрічі, наприклад, Nexus Sprint Planning, де координується спільна робота над продуктом. Всі Скрам команди в Nexus мають єдиний Product Backlog, відповідно єдиного Product Owner.

Серед аналогів Nexus можна виділити **Scaled Agile Framework (SAFe)**, який підходить для великих організацій із жорсткою ієрархією, та **Large-Scale Scrum (LeSS)**, який більше сфокусований на гнучкості та мінімальній бюрократії. Також існує **Spotify Model**, яка відрізняється гнучким підходом до структуризації команд у “племена” (tribes) і не прив'язана жорстко до Scrum. На відміну від них, Nexus зберігає максимально традиційний Scrum, додаючи тільки необхідні елементи для інтеграції. Це робить його ідеальним для організацій, які вже працюють за Scrum і хочуть масштабуватися без значних змін. Nexus описаний в Nexus Guide (2021, 10 pages).

Ролі в команді та модель компетенції

Розглянемо розподіл ролей в плоских та ієрархічних командах. У Скрамі та XP команди плоскі, але оскільки на практиці багато хто працює в ієрархічних командах, ми розглянемо можливий розподіл ролей в них.

Принцип Collective Ownership у Scrum і XP означає, що вся команда несе спільну відповідальність за результат роботи. У Scrum це проявляється через колективну відповідальність за інкремент продукту, відкриту комунікацію й відсутність жорсткого розподілу ролей. У XP цей принцип підкреслюється можливістю кожного змінювати будь-який код, парним програмуванням (pair programming) і спільним дотриманням стандартів (code style guides), що підвищує якість та прозорість роботи.

У Scrum команді відсутність жорсткої ієрархії сприяє мінімізації диктатури та дозволяє уникнути концентрації влади в одних руках. Всі учасники мають рівні права на прийняття рішень, що забезпечує загальне залучення до процесу. Це створює атмосферу, де кожен може вносити свої ідеї та бути почутим, що підвищує ефективність співпраці та дозволяє приймати більш зважені і різнобічні рішення.

Також Scrum підтримує принцип спільної відповідальності, де команда працює разом над досягненням загальної мети. Кожен член команди, незалежно від ролі, відповідає за кінцевий результат, що створює атмосферу взаємопідтримки і взаємної довіри. Цей підхід дозволяє забезпечити кращу якість роботи, зменшити ризики і швидше реагувати на зміни в умовах проєкту.

В Scrum рішення не приймаються через голосування, а досягаються через консенсус, подібно до процесу у суді присяжних. Це означає, що команда прагне до узгодженого рішення, де кожен має можливість висловити свою думку та врахувати позицію інших. Такий підхід забезпечує, що рішення приймаються на основі колективного обговорення, що дозволяє знайти найкращий шлях для вирішення проблеми і врахувати різні точки зору.

Консенсус дозволяє уникнути конфліктів і створює атмосферу співпраці, де кожен член команди відчуває свою причетність до прийнятого рішення. Це також сприяє досягненню спільної відповідальності за кінцевий результат, оскільки всі учасники активно беруть участь у процесі, а не покладаються на волю однієї особи чи групи.

Часто практикують ієрархічні команди.

Модель ієрархічної команди за Фредом Бруксом з “Міфічний людино-місяць”:
- Головний програміст-архітектор
- Старший розробник
- Бізнес-аналітик, він же редактор документації
- Тестувальник
- Системний адміністратор

Ось приклад ролей в ієрархічній команді, які пропонує Feature-driven Development (FDD, 1997, тобто “Розробка, керована функціональністю”).
FDD визначає шість ключових ролей і передбачає низку інших.

- Керівник проєкту (Project Manager) є адміністративним керівником проєкту, відповідальним за звітування про прогрес, управління бюджетами, боротьбу за кількість персоналу, управління обладнанням, приміщенням та ресурсами тощо.
- Головний архітектор (Chief architect) відповідає за загальний дизайн системи. Він відповідає за проведення дизайн-сесій, на яких команда співпрацює у розробці системи. Ця робота вимагає як відмінних технічних та моделюючих навичок, так і хороших навичок фасилітації. Він або вона керує проєктом через технічні перешкоди, з якими стикається команда.

- Менеджер розробки (Development manager) відповідає за керівництво щоденною розробкою. У ролі фасилітатора, яка вимагає хороших технічних навичок, менеджер розробки відповідає за вирішення щоденних конфліктів щодо ресурсів, якщо головні програмісти не можуть зробити це між собою.

- Головні програмісти (Chief programmers) – це досвідчені розробники, які вже кілька разів проходили через весь життєвий цикл розробки програмного забезпечення. Вони беруть участь у аналізі вимог на високому рівні та в проєктуванні, а також керують невеликими командами з трьох-шести розробників у процесі низькорівневого аналізу, проєктування та розробки нових функцій програмного забезпечення.

- Власники класів (Class owners) – це розробники, які працюють у складі невеликих команд розробки під керівництвом головного програміста, займаючись проєктуванням, програмуванням, тестуванням та документуванням функцій, необхідних для нової системи програмного забезпечення.

- Експерти домену (Domain experts) – це користувачі, стейкхолдери, спонсори, бізнес-аналітики або будь-яке поєднання цих ролей. Вони є базою знань, на яку спираються розробники для створення правильної системи. Експерти домену повинні мати хороші навички усного, письмового та презентаційного спілкування. Їхні знання та участь є абсолютно критичними для успіху системи, що створюється.

Процес початку проєкту розпочинається з найму команди, який здійснює керівник проєкту (Project manager). Він визначає бюджет і ресурси, необхідні для реалізації проєкту, а також відповідає за звітність про прогрес. Керівник проєкту обирає головного архітектора (Chief architect), який відповідатиме за загальний дизайн системи. Головний архітектор проводить дизайн-сесії, де команда спільно розробляє технічні рішення.
Після визначення дизайну головний архітектор делегує завдання менеджеру розробки (Development manager), який організовує повсякденну діяльність команди. Менеджер розробки вирішує щоденні конфлікти, координує роботу та залучає головних програмістів до процесу. Головні програмісти (Chief programmers), у свою чергу, керують невеликими командами власників класів (Class owners), які займаються проєктуванням, програмуванням та тестуванням нових функцій.
В цей час експерти домену (Domain experts), які можуть бути користувачами, стейкхолдерами або бізнес-аналітиками, надають необхідні знання та інформацію, критично важливі для розробки системи. Вони забезпечують зв'язок між командою розробки та кінцевими користувачами, допомагаючи уточнити вимоги та перевірити їхню відповідність. Цей структурований процес делегування ролей та відповідальностей сприяє ефективному управлінню проєктом і забезпечує досягнення поставлених цілей.
Ще в FDD є принцип Individual Class (Code) Ownership, тобто Індивідуальне володіння класом (кодом), що означає закріплення конкретного шматка коду за конкретним розробником. Тобто інший розробник не може змінювати чужий код.

Власник класу є єдиною особою, якій дозволено вносити прямі зміни до свого класу. Інші розробники можуть пропонувати зміни, але власник повинен їх схвалити та впровадити.
Це створює структурований робочий процес, де власність чітка, а обов'язки чітко визначені.
FDD пропонує ієрархічну модель, яка несе всі її мінуси та плюси.

Модель компетенції SWECOM

Крім ролі є ще рівень компетенції.

Модель компетенції SWECOM, крім загальних компетенції вводить компетенції по конкретним навичкам та вмінням.
Модель компетенцій у галузі програмної інженерії (SWECOM, IEEE) організована за областями навичок (наприклад, вимоги до програмного забезпечення), навичками в межах цих областей (наприклад, збір вимог до програмного забезпечення) та діяльністю в межах навичок (наприклад, створення прототипу для збору вимог).
Діяльність класифікується за п'ятьма рівнями компетенцій:
• Технік (Technician)
• Практик початкового рівня (Entry Level Practitioner)
• Практик (Practitioner)
• Технічний керівник (Technical Leader)
• Старший інженер-програміст (Senior Software Engineer)

Загалом, Технік виконує інструкції, Практик початкового рівня допомагає у виконанні діяльності або виконує її під наглядом; Практик виконує діяльність з мінімальним або без нагляду; Технічний керівник керує окремими людьми та командами під час виконання діяльності; Старший інженер-програміст змінює існуючі методи та інструменти, а також створює нові. Деякі організації можуть об'єднувати рівні Техніка і Практика початкового рівня. Старший інженер-програміст може виконувати роль "головного інженера" (CTO) для організації з розробки програмного забезпечення, і деякі з них можуть бути визнані експертами галузі, які роблять внесок у формування та розвиток професії програмної інженерії.
Крім діяльності, визначеної на різних рівнях компетенції, додатковою компетенцією всіх інженерів-програмістів є навчання та наставництво інших, у відповідних випадках, у методах, інструментах і техніках, які використовуються для виконання цих діяльностей.
Наприклад, Технік або Практик початкового рівня можуть навчати або наставляти інших у використанні інструментів керування конфігурацією для виконання своїх завдань, або Технічний керівник може навчати чи наставляти Практика в тому, як проводити інспекції та огляди.

Дисфункції команди

-Мікроменеджмент

-Диктатура
-Недовіра

Мікроменеджмент виникає, коли керівник надмірно контролює кожен аспект роботи команди, позбавляючи її автономії. Це знижує ініціативність працівників, викликає стрес і відволікає менеджера від стратегічних завдань. В результаті команда стає менш продуктивною та втрачає мотивацію.

Диктатура в управлінні проявляється у прийнятті рішень без врахування думок команди. Такий підхід створює відчуження, знижує рівень залученості співробітників і обмежує їхню креативність. Команда втрачає відчуття впливу на процеси, що призводить до втрати довіри до керівника.
В книзі “Як пасти котів” Генк Рейнвотер писав: “Диктатор намагається нав’язувати своїм підлеглим конкретні методи вирішення поставлених перед ними завдань. Насправді всім нам слід ретельно стежити за тим, щоб не піти цим шляхом. Лідерство дійсно передбачає демонстрацію співробітникам можливих способів роботи, але тут важливо бути обережним, адже позбавляти підлеглих відчуття причетності до створення продукту неприпустимо”.

Недовіра у команді або між працівниками та керівником руйнує взаємодію. Вона проявляється у небажанні ділитися ідеями, конфліктах і відсутності співпраці. У результаті команда перестає працювати як цілісний механізм, а продуктивність падає.

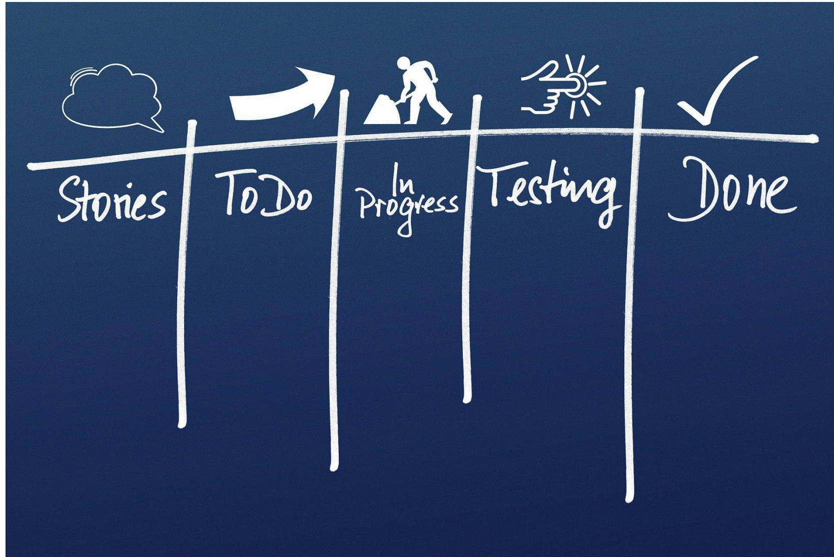
Щоб усунути ці дисфункції, необхідно створити культуру довіри, делегувати повноваження, заохочувати відкриту комунікацію і залучати команду до прийняття рішень. Такі дії сприятимуть розвитку відповідальності, мотивації та взаємної підтримки.

У Scrum розробники самостійно визначають, як перетворити елементи Product Backlog на Інкременти, які відповідають Definition of Done. Це означає, що тільки вони мають право вирішувати, як розбити завдання та організувати роботу для досягнення результату. Жоден інший член команди, включаючи Product Owner чи Scrum Master, і тим більше сторонні особи, не можуть втручатися в цей процес.

Такий підхід базується на принципі самоорганізації та підвищує відповідальність розробників за виконану роботу. Це також сприяє мотивації команди, оскільки їй надається повна автономія в ухваленні технічних рішень, необхідних для створення цінного Інкремента.

User stories

Приклад канбан дошки



Критерій INVEST

В гнучких (agile) методологіях розробки бізнес-аналітик дуже близький до ролі Продукт овнера.
Бізнес-аналітик в гнучких методологіях пише тільки ту документацію, без якої ніяк, все інше вважається надлишковим. Definition of Done обговорюється всією Agile командою.

Абревіатура INVEST допомагає запам'ятати характеристики хороших історій (user stories) в Agile:
I – Незалежна (Independent): Історії повинні бути максимально незалежними.
N – Переговорна (Negotiable): Деталі можуть уточнюватися в процесі.
V – Цінна (Valuable): Історія повинна приносити користь користувачам.
E – Оцінювана (Estimable): Має бути можливість оцінити зусилля на реалізацію.
S – Невелика (Small): Історія повинна бути достатньо маленькою для завершення в одному спринті.
T – Тестована (Testable): Історія повинна мати чіткі критерії прийняття.

Як писати вимоги бізнес-аналітику:
0. Пояснити структуру вимог всім (принцип опису вимог).
1. Загальний опис зробити перед описом деталей.
2. Орієнтація на тест кейси, а не на розлогі формулювання.
3. Писати як для тупих, тобто повну інфу давати, щоб не прийшлося додумувати.
KISS - Keep it simple, stupid.
4. Підкріплювали вимоги скріншотами, якщо можливо.
5. Не писати технічні нюанси, за це відповідають розробники.
6. Вимогу перевіряє команда.
7. Не роздувати опис завдання, робити декомпозицію на декілька юзер сторі.

Agile чи Waterfall?

Agile цілком підходить для великих, складних і критично важливих проєктів, проте, якщо у вас виконуються наведені нижче умови, тоді Waterfall (або Spiral) та ієрархічна команда можуть мати переваги, а саме коли:

- Чітко визначені вимоги

Якщо всі вимоги до проєкту заздалегідь відомі, стабільні і не змінюватимуться протягом усього життєвого циклу. Наприклад, в урядових чи регуляторних проєктах, де необхідно суворо дотримуватись стандартів.

- Фіксований бюджет та строки

Коли замовник потребує чіткого прогнозу витрат і часових рамок, а зміни у процесі розробки неприйнятні.

- Пріоритет на документацію

Якщо проєкт вимагає великої кількості детальної документації, яка буде використовуватись на різних етапах (наприклад, у великих інженерних або виробничих проєктах).

- Немає потреби у частих перевірках або зворотному зв’язку

Якщо замовник не хоче (або не може) брати активну участь у процесі розробки, і затвердження відбувається лише в кінці.

- Критичні системи

У розробці програмного забезпечення для систем, де недоліки можуть призвести до серйозних наслідків (медичне обладнання, авіація, оборонна галузь).

Однак загалом гнучкі методології мають перевагу.

CI/CD та Git

1978 року Стівен Кертіс Джонсон, науковець комп'ютерної галузі з лабораторії Белла, написав програму lint.
Лінтер — це інструмент для аналізу коду, який допомагає знаходити потенційні помилки, проблеми зі стилем коду та інші недоліки.
Лінтер не вводить нові синтаксичні конструкції, а просто аналізую програму.
Синтаксичні конструкції та їх семантика вводяться на рівні мови.
Лінтери зараз включають в CI/CD pipeline.

Concurrent Versions System (CVS) — система керування версіями, спочатку розроблена Діком Груном у липні 1986 року.
CollabNet заснував проєкт Subversion у 2000 році як спробу написати систему керування версіями з відкритим вихідним кодом, яка працювала подібно до CVS, але виправила помилки та надала деякі функції, відсутні в CVS.

Сьогодні Git є де-факто стандартною системою контролю версій. Лінус Торвальдс почав розробляти Git у квітні 2005 року після того, як безкоштовну ліцензію на BitKeeper, пропрієтарну source-control management (SCM) систему, яка використовувалася для розробки ядра Linux з 2002 року, було відкликано для Linux.

Matt Mackall вперше анонсував розподілену систему керування версіями Mercurial 19 квітня 2005 року. Поштовхом до цього стало оголошення Bitmover про те, що вони припиняють доступ до безкоштовної версії BitKeeper. Цей проєкт розпочався через кілька днів після того, як Лінус Торвальдс ініціював добре відомий проект Git з подібними цілями.

Приблизно в 2009 з'явилася trunk-based model для керування гілками.

В 2010 році Vincent Driessen презентував модель GitFlow.

Екстремальне програмування та DevOps практики наголошують на застосуванні CI/CD (Continuous integration and Continuous delivery).

Неперервна інтеграція — це практика, а не інструмент. ЇЇ чітко описав й використовував Кент Бек в 1999 році.

У 2001 році ThoughtWorks істотно змінила гру. Вони створили CruiseControl, перший інструмент неперервної збірки.
Неперервна збірка (continuous build) ніколи не повинна ламатися.
У 2008 році Hudson став популярною альтернативою CruiseControl та іншим серверам збірки з відкритим вихідним кодом. Після заміни на Jenkins Hudson більше не підтримується.

Неперервна доставка (Continuous delivery, CD) — це практика розробки програмного забезпечення, за якої зміни в коді автоматично готуються для випуску в продакшн. Це одна з ключових основ сучасної розробки застосунків. Неперервна доставка розширює концепцію неперервної інтеграції, автоматично впроваджуючи всі зміни в коді до тестового середовища та/або продакшну після етапу збірки. За умови правильної реалізації розробники завжди мають готовий до розгортання артефакт збірки, що пройшов стандартизований процес тестування.

Неперервна інтеграція (Continuous Integration, CI) - це практика в розробці програмного забезпечення, коли код розробляється інкрементально та регулярно інтегрується в

спільний репозиторій. Це допомагає виявляти конфлікти та помилки якомога раніше в розробці, забезпечуючи стабільніше та надійніше програмне забезпечення.
Неперервна інтеграція — це практика, а не інструмент. Її чітко описав й використовував Кент Бек в 1999 році.

Нарис CI/CD з trunk-based model

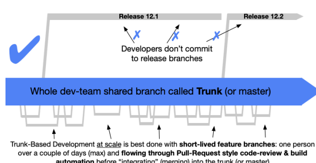
CI/CD процес починається з вибору платформи для зберігання та управління кодом. Репозиторії в GitHub або GitLab виступають центральним місцем для командної роботи. Основна гілка (main або master) слугує базовою (trunk) для інтеграції змін, а розробники працюють у короткоживучих гілках, створюючи функціонал або виправляючи помилки. Після завершення роботи зміни проходять код-рев'ю, де використовується інструменти для спільного огляду коду, наприклад, GitLab Merge Requests або GitHub Pull Requests.

Перед злиттям змін в гілку main виконуються автоматичні тести та аналіз коду лінтерами, які є частиною pipeline. Husky може встановлювати pre-commit хуки, щоб запускати локальні тести або лінтер перед внесенням змін у репозиторій. Для стилю коду команда обирає стандарт, наприклад, Airbnb JavaScript Style Guide. Це забезпечує єдиний формат коду, який легко підтримувати.

Процес інтеграції коду плавно переходить у неперервне розгортання. Автоматизація деплойменту (розгортання) досягнута за допомогою платформ, таких як AWS App Runner або DigitalOcean App Runner. Docker контейнери використовуються для стандартизації середовища розробки та розгортання, що робить процес більш передбачуваним. При кожному оновленні репозиторію оновлений код автоматично розгортається на сервері.

Для управління процесом розробки використовуються інструменти Agile, наприклад, Jira або Trello, з Kanban-дошкою. Команда працює за Scrum фреймворком або Kanban методом, що дозволяє планувати релізи, використовуючи feature flags для поступового впровадження функціоналу. Перед кожним релізом створюється окрема release-гілка, яка проходить фінальне тестування перед розгортанням на продакшн.

В основі процесу лежить trunk-based підхід. Розробники працюють з гілкою main, обмежуючи використання feature branches до коротких ітерацій. Це гарантує, що код інтегрується з основною гілкою щонайменше раз на 16 годин. Релізи залишаються керованими, оскільки новий функціонал активується тільки після ретельного тестування.



(Image from https://trunkbaseddevelopment.com/)

Джерела

- "The Mythical Man-Month" (1975), Fred Brooks.
- "Becoming a Technical Leader" (1986), Gerald Weinberg.
- “The Machine That Changed the World” (1990), James P. Womack, Daniel T. Jones, and Daniel Roos.
- Project Management Body of Knowledge (PMBOK), 1996.
- “Lean Thinking: Banish Waste and Create Wealth in Your Corporation” (1996), James P. Womack, Daniel T. Jones.
- "Extreme Programming Explained" (1999), Kent Beck.
- "Agile Software Development: Principles, Patterns, and Practices" (2002), Robert C. Martin.
- "A Practical Guide to Feature-driven Development" (2002), Stephen R. Palmer.
- "Herding Cats: A Primer for Programmers Who Lead Programmers" (2002), Hank Rainwater.
- "The Five Dysfunctions of a Team" (2002), Patrick Lencioni.
- “Crystal Clear: A Human-Powered Methodology for Small Teams” (2004), Alistair Cockburn.
- "Agile Estimating and Planning" (2005), Mike Cohn.
- “Agile Testing: A Practical Guide for Testers and Agile Teams” (2008), Lisa Crispin.
- "Scrum Guide" (2010), Ken Schwaber, Jeff Sutherland.
- “Agile product management with Scrum: creating products that customers love” (2010), Roman Pichler.
- "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" (Addison-Wesley, 2010), Jez Humble, David Farley.

- "Continuous Integration: Improving Software Quality and Reducing Risk" (Addison-Wesley, 2010), Paul M. Duvall, Steve Matyas, Andrew Glover.

- "Essential Scrum: A Practical Guide to the Most Popular Agile Process" (2012), Kenneth Rubin.

- “Software Requirements” (2013), Karl Wiegers.

- "Scrum: The Art of Doing Twice the Work in Half the Time" (2014), Jeff Sutherland.

- "SWECOM" (2014), IEEE.

- "Learning Agile: Understanding Scrum, XP, Lean, and Kanban" (2014), Andrew Stellman.

- The Nexus Guide (2015).

- "Agile Project Management For Dummies" (2017), Mark C. Layton.

- "The Manager’s Path: A Guide for Tech Leaders" (2017), Camille Fournier.

- "Clean Agile: Back to Basics" (2019), Robert Martin.

- “Staff Engineer: Leadership Beyond the Management Track” (2021), Will Larson.

- “The Staff Engineer's Path: A Guide for Individual Contributors Navigating Growth and Change” (2022), Tanya Reilly.

- https://wiki.c2.com/?ExtremeProgramming
- https://www.scrum.org/
- https://medium.com/@dmytro1popov/%D1%82%D1%80%D0%B8-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%96-%D0%BE%D1%80%D0%B3%D0%B0%D0%BD%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%97-%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B8-4962935886d7